**🧩 What is @PutMapping?**

**@PutMapping** is an **annotation** in Spring Boot that is used to **handle HTTP PUT requests**.

👉 In RESTful web services,

* **PUT** is a type of HTTP method (like GET, POST, DELETE).
* It is used when you want to **update** an **existing resource** (like updating a record in a database).

**🧠 Simple Meaning**

Think of it like this:

If **@PostMapping** is used to “create something new”,  
then **@PutMapping** is used to “update something that already exists”.

**📦 Example Scenario**

Let’s say we have a **Student Management System**, and we want to **update a student’s details** (like name or email) based on their ID.

**🧑‍💻 Example Code**

package in.orcas.controller;

import org.springframework.web.bind.annotation.\*;

import in.orcas.model.Student;

@RestController

@RequestMapping("/students")

public class StudentController {

// Update existing student details

@PutMapping("/{id}")

public String updateStudent(@PathVariable Integer id, @RequestBody Student student) {

// Here we assume that student details are updated in the database

return "Student with ID " + id + " has been updated successfully!";

}

}

**🧩 Explanation of Each Part**

**1. @PutMapping("/{id}")**

* This tells Spring Boot:  
  “If a **PUT** request comes to the path /students/{id}, this method should handle it.”
* The {id} part means it will accept any ID value dynamically — e.g., /students/5.

**2. @PathVariable Integer id**

* It reads the **ID** from the **URL path**.  
  Example: If the request URL is /students/5, then id = 5.

**3. @RequestBody Student student**

* It reads the **updated student data** (in JSON format) from the **request body** and converts it into a **Student Java object**.
* Example JSON sent from the client:
* {
* "name": "Ravi Kumar",
* "email": "ravi@gmail.com"
* }

**4. Inside the Method**

* Normally, we find that student in the database (using id).
* Then we replace or update the old values with the new ones from the request body.
* Finally, we save the updated record.

**🌐 Example Request**

**🔹 URL:**

PUT http://localhost:8080/students/5

**🔹 Request Body (JSON):**

{

"name": "Pavan Kalyan",

"email": "pavan@gmail.com"

}

**🔹 Response:**

Student with ID 5 has been updated successfully!

**⚙️ When to Use @PutMapping**

| **Situation** | **Use** |
| --- | --- |
| You want to create a **new record** | @PostMapping |
| You want to **update the full record** | @PutMapping |
| You want to **update only part of a record** | @PatchMapping |
| You want to **delete** a record | @DeleteMapping |

**⚖️ PUT vs POST (Important Difference)**

| **Feature** | **POST** | **PUT** |
| --- | --- | --- |
| Purpose | Create new resource | Update existing resource |
| Idempotent (same request multiple times → same result?) | ❌ No | ✅ Yes |
| Example | Add new student | Update existing student |

💡 “Idempotent” means — sending the same PUT request 5 times won’t change the result after the first time.  
But POST will keep creating new entries each time.

**🧭 Real-Life Analogy**

Imagine an **online form** for your student profile:

* 📝 **POST** → when you **submit** the form for the first time to **create your profile**.
* 🧰 **PUT** → when you **edit** your profile (like updating your email or name).
* Each PUT replaces your **old details** with **new ones**.

**✅ Summary**

| **Concept** | **Description** |
| --- | --- |
| Annotation | @PutMapping |
| Purpose | To update existing data on the server |
| HTTP Method | PUT |
| Common Use | Update record (e.g., update student, product, user) |
| Typical Input | Path variable (id) + Request body (JSON) |
| Response | Usually confirmation message or updated object |

**🧩 The Core Idea**

👉 The **frontend’s job** is to **send data** to the backend.  
👉 The **backend’s job** is to **receive and process that data**.

Now, **how the data is sent** depends on **how the frontend code is written.**  
There are **two main ways** the frontend can send data to the backend:

**🧠 1. Sending data in the URL**

Used mostly with @PathVariable or @RequestParam

Example:

GET http://localhost:9090/api/student/101

Backend:

@GetMapping("/student/{id}")

public Student getStudent(@PathVariable int id) {

// id will be 101

}

➡️ Here, data travels **inside the URL**.  
It’s visible in the browser or network tab — useful for simple or small data like IDs, names, or filters.

**🧠 2. Sending data in the Request Body (JSON)**

Used with @RequestBody

Example:  
Frontend sends:

{

"id": 101,

"name": "Sachin Tendulkar",

"email": "sachin@gmail.com"

}

Backend:

@PostMapping("/save")

public String saveCustomer(@RequestBody Customer customer) {

// Spring converts JSON → Java Object automatically

return "Saved " + customer.getName();

}

➡️ Here, data travels **inside the request body** (not visible in URL).  
This is used when you send **large or complex data**, like forms, objects, or nested JSON.

**⚖️ So, in short:**

| **Data Sending Method** | **Used In** | **Annotation** | **Example** | **Use Case** |
| --- | --- | --- | --- | --- |
| URL (path) | GET, DELETE | @PathVariable | /student/101 | Small data (like IDs) |
| URL (query) | GET | @RequestParam | /student?id=101 | Filters or search |
| JSON Body | POST, PUT | @RequestBody | { "id":101, "name":"Sachin" } | Big data (form submission) |

**🧩 In Real-time Projects**

✅ You’ll mostly use

* @RequestBody for **POST/PUT** (sending object data)
* @PathVariable or @RequestParam for **GET/DELETE** (fetching/deleting specific data)

Eg: PutMapping

**🧩 What is @PatchMapping?**

@PatchMapping is an annotation in **Spring Boot** used to handle **HTTP PATCH requests**.

It’s similar to @PutMapping, but the **main difference** is:

👉 PUT = Replace the entire object  
👉 PATCH = Update only part of the object

**🧠 In simple words**

| **Action** | **Meaning** |
| --- | --- |
| @PutMapping | “Replace the whole record with this new data.” |
| @PatchMapping | “Change only the specific fields you mention.” |

**🏥 Analogy**

Think of a **student record** as a file in a drawer.

* **PUT** → You throw away the old file and put a new one in its place.
* **PATCH** → You open the old file and correct just a few lines (like email or phone).

**🧪 Example Scenario**

You have a student record in DB:

{

"id": 101,

"name": "Pavan",

"email": "pavan@gmail.com",

"city": "Hyderabad"

}

**🧩 Case 1 — Using PUT**

If you send:

{

"name": "Pavan Kumar",

"email": "pavank@gmail.com"

}

With PUT, the backend expects **all fields** — if not provided, missing ones may get replaced with null (depending on code).  
So, the record could become:

{

"id": 101,

"name": "Pavan Kumar",

"email": "pavank@gmail.com",

"city": null // Lost! Because PUT replaced everything

}

**🧩 Case 2 — Using PATCH**

If you send the same:

{

"email": "pavank@gmail.com"

}

With PATCH, only the email field is updated,  
other fields (name, city) remain unchanged. ✅

Final record:

{

"id": 101,

"name": "Pavan",

"email": "pavank@gmail.com",

"city": "Hyderabad"

}

**⚙️ Spring Boot Example**

**Controller**

@RestController

@RequestMapping("/students")

public class StudentController {

@PatchMapping("/{id}")

public String updatePartialStudent(

@PathVariable Integer id,

@RequestBody Map<String, Object> updates) {

// Here, we receive only partial data as a Map

// Example: { "email": "newmail@gmail.com" }

// 1️⃣ Fetch existing student from DB

// Student student = studentService.getStudentById(id);

// 2️⃣ Apply updates manually

// if (updates.containsKey("email"))

// student.setEmail((String) updates.get("email"));

// if (updates.containsKey("city"))

// student.setCity((String) updates.get("city"));

// 3️⃣ Save back to DB

// studentService.save(student);

return "Student " + id + " partially updated successfully!";

}

}

**Request Example (Frontend → Backend)**

**URL:**

PATCH http://localhost:8080/students/101

**Body (JSON):**

{

"email": "pavank@gmail.com"

}

✅ Only email field will be updated.

**🧭 Important Notes**

| **Concept** | **Description** |
| --- | --- |
| @PatchMapping | Used for **partial updates** |
| @PutMapping | Used for **complete replacement** |
| HTTP Method | PATCH |
| Backend Parameter | Usually @RequestBody Map<String,Object> or DTO |
| Common Usage | Update only 1 or 2 fields (like email, phone, address, etc.) |
| Frontend Input | Usually JSON body (same as PUT) |

**⚙️ When to Use PUT vs PATCH**

| **Situation** | **Preferred Method** | **Example** |
| --- | --- | --- |
| Updating all fields | PUT | Update full profile |
| Updating only a few fields | PATCH | Change just email or address |
| Replacing resource completely | PUT | Replace full object |
| Correcting small details | PATCH | Update partial data |

**🧩 Example Comparison Summary**

| **Aspect** | **PUT** | **PATCH** |
| --- | --- | --- |
| Purpose | Full update | Partial update |
| Required data | All fields | Only changed fields |
| Safe for missing fields | ❌ Might nullify | ✅ Keeps old values |
| Backend annotation | @PutMapping | @PatchMapping |
| Typical frontend request | JSON in body | JSON in body |

**⚙️ Real-Time Practice in Frontend**

Frontend code is **almost same as PUT**, only the HTTP method changes:

fetch("http://localhost:8080/students/101", {

method: "PATCH",

headers: {

"Content-Type": "application/json"

},

body: JSON.stringify({ email: "pavank@gmail.com" })

});

**🧠 Quick Recap**

| **Question** | **Answer** |
| --- | --- |
| What is @PatchMapping? | Used to handle HTTP PATCH requests for partial updates. |
| When to use it? | When you only want to modify specific fields, not replace the whole object. |
| What does backend receive? | JSON data (partial fields only). |
| What’s the frontend difference? | Same as PUT, but uses method: "PATCH". |

**✅ Final Summary**

| **Feature** | **@PutMapping** | **@PatchMapping** |
| --- | --- | --- |
| HTTP Method | PUT | PATCH |
| Update Type | Complete | Partial |
| Input JSON | Full object | Only changed fields |
| Common Use | Replace record | Update one or two fields |
| Real-time use | Less frequent (for full update) | More frequent (for partial form edits) |

Eg: PatchMapping

## 🧩 Your Question in Simple Terms:

“I sent JSON data, so I set Content-Type: application/json.  
But the response I got was just plain text (not JSON).  
So why did we still set Accept: application/json?”

# 🧠 1️⃣ What Each Header Means (Quick Recap)

| **Header** | **Who Sets It** | **What It Means** |
| --- | --- | --- |
| **Content-Type** | Client (Postman/Frontend) | “I’m sending data in this format” |
| **Accept** | Client (Postman/Frontend) | “I want the response back in this format” |

So:

* Content-Type → describes **your request body**
* Accept → requests **your preferred response format**

# ⚙️ 2️⃣ What Happened in Your Example (and why you got plain text)

In your controller, the method returns **String**, like this:

@PostMapping

public String createStudent(@RequestBody Student student) {

Student saved = service.saveStudent(student);

return "✅ Student created with ID: " + saved.getId();

}

That means you’re **sending text (not JSON)** back.

✅ So, even though you wrote  
Accept: application/json,  
Spring Boot looked at your method’s return type and said:

“This method is returning text, not JSON. So I’ll send plain text.”

That’s perfectly fine — Spring Boot respects the method’s return type over the Accept header.

# 🧩 3️⃣ So… Was Accept: application/json wrong here?

No, it wasn’t wrong.  
It just wasn’t **used** — because the controller didn’t return JSON.

You can still keep it — it’s a **good practice** because:

1. It’s consistent with other API calls (like GET, which return JSON).
2. It doesn’t cause any issue.
3. It makes Postman behave consistently for your whole API collection.

# 🧠 4️⃣ When Accept: application/json Actually Matters

It matters **only** when your backend **can return multiple formats** —  
and you’re telling the server which one you want.

Example:

@GetMapping(value = "/{id}", produces = {"application/json", "application/xml"})

public Student getStudent(@PathVariable Integer id) {

return service.getStudent(id);

}

Now if you set:

| **Accept Header** | **Response Format** |
| --- | --- |
| application/json | You’ll get JSON ✅ |
| application/xml | You’ll get XML ✅ |

This is called **content negotiation** — the client “negotiates” with the server about the response format.

# 🧩 5️⃣ Let’s Fix It to Return JSON (if you want)

If you want your POST response also to return **JSON**,  
just return the Student object instead of text.

@PostMapping

public Student createStudent(@RequestBody Student student) {

return service.saveStudent(student);

}

Now, when you send:

**Request Headers:**

Content-Type: application/json

Accept: application/json

**Response will be:**

{

"id": 101,

"name": "Pavan",

"email": "pavan@gmail.com",

"city": "Hyderabad"

}

✅ Now your Accept: application/json header is meaningful —  
Spring Boot returns JSON because your method returns a Java object.

# 🧭 6️⃣ Summary

| **Header** | **You use it when...** | **Effect** |
| --- | --- | --- |
| Content-Type: application/json | You’re **sending** JSON data (POST, PUT, PATCH) | Tells Spring to **parse** body into Java object |
| Accept: application/json | You **expect JSON** in response | Tells Spring to **return JSON** if possible |
| You got text | Because method returned a String | Spring sent text response |

✅ **Final Takeaway:**

You must always use Content-Type: application/json when sending JSON.  
You may use Accept: application/json for consistency — it only takes effect if the server returns an object (not plain text).